C Programming

Operating SystemsVU 2023W

Florian Mihola, David Lung, Andreas Brandstätter,
Axel Brunnbauer, Peter Puschner

Technische Universität Wien
Computer Engineering
Cyber-Physical Systems

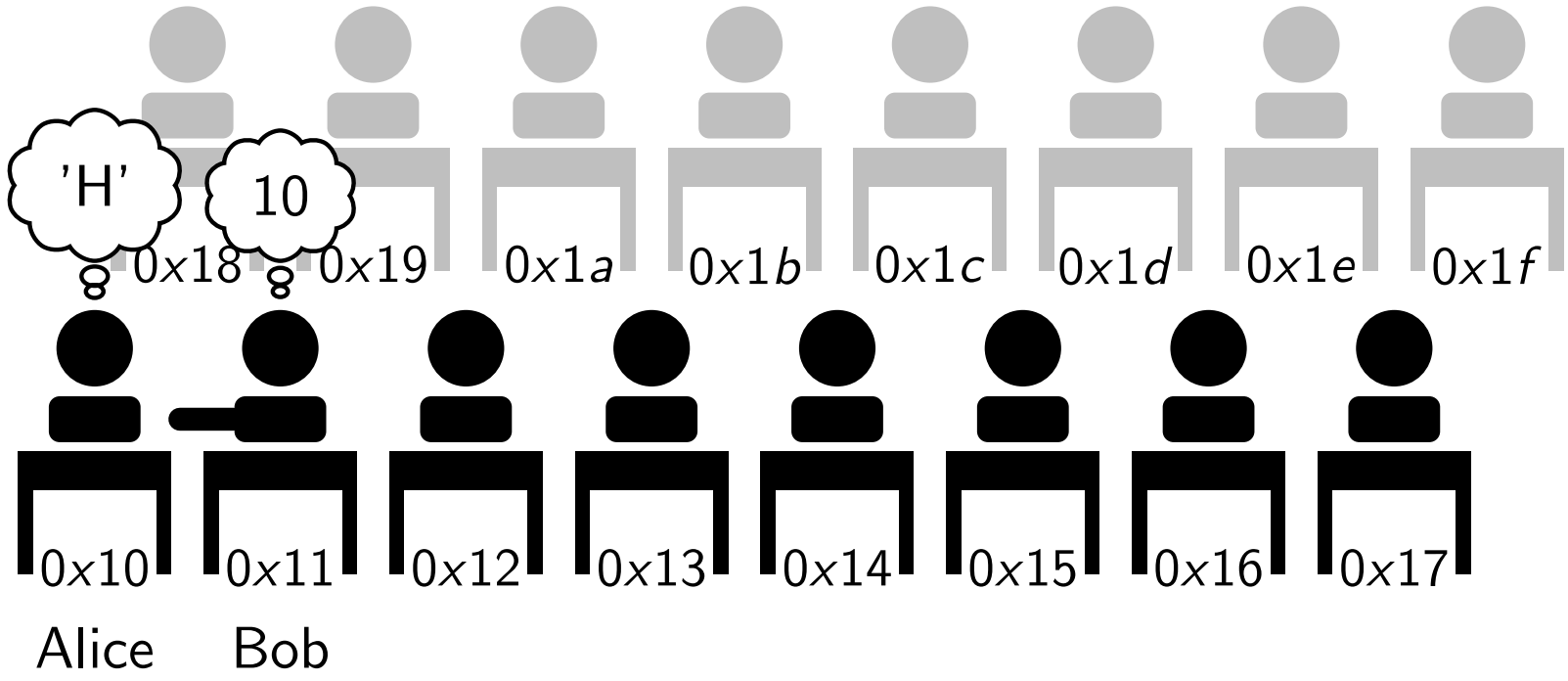
2023-10-05

Content

- Pointer
- Complex data structures
  - void
  - struct
  - union
  - enum
  - Nesting
  - typedef
  - Alignment
- Functions
  - Parameters
  - Return values
  - inline
  - Pointers to functions
- Procedural programming
- Modular programming

Pointer

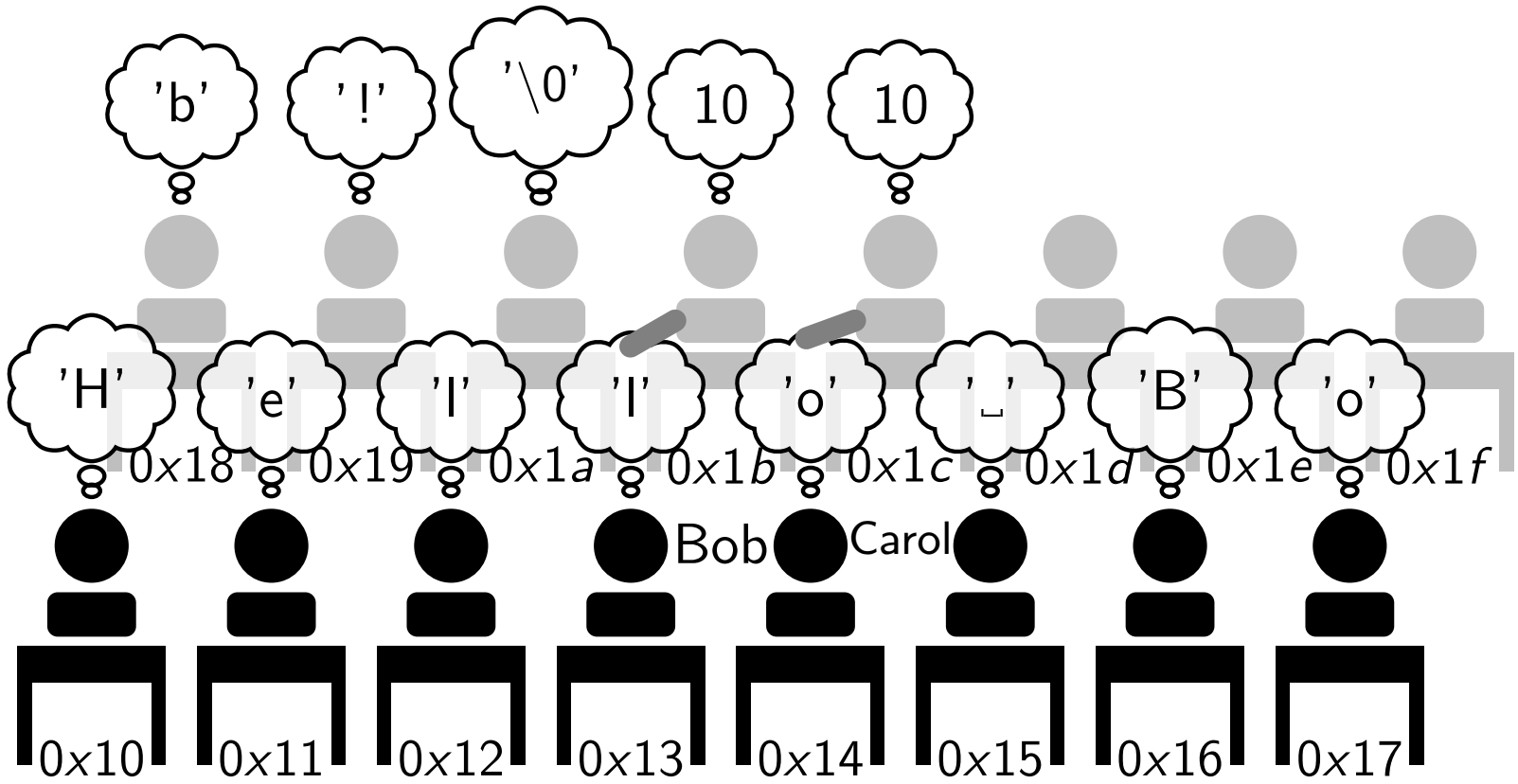
```
char alice = 'H';
char *bob = &alice; /* alice' addr: 10 */
```




```
printf("%x", alice);   /* prints 48 (hex for 'H')   */
printf("%x", bob);     /* prints 10 */
printf("%x", &alice);  /* prints 10 */
printf("%x", &bob);    /* prints 11 */
printf("%x", *bob);    /* prints 48 (hex for 'H')   */
```

Pointer

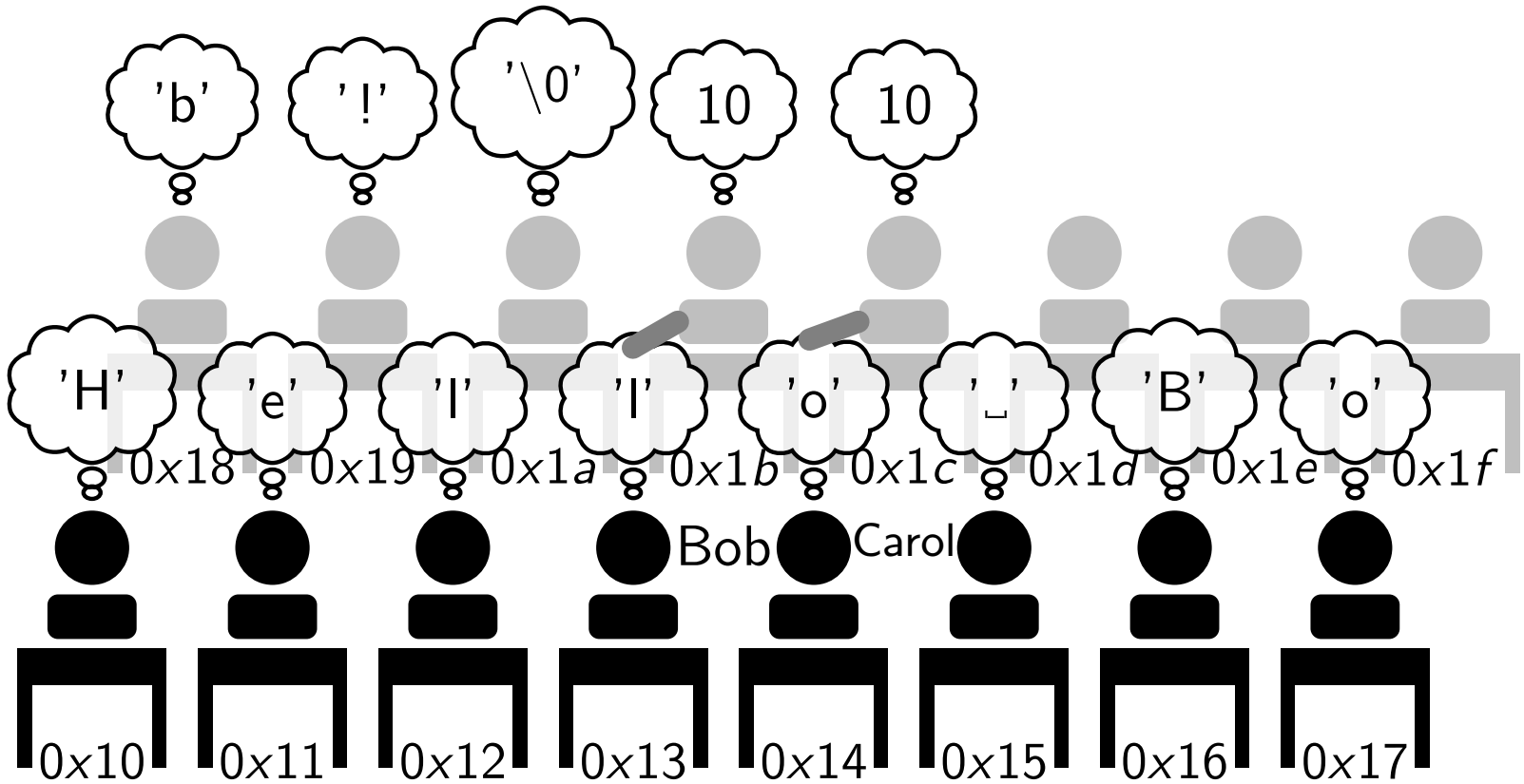

```c
char  alice[] = "Hello Bob!";/* auto \0 terminated */
char  *bob    = &alice;      /* alice' addr: 10 */
short *carol  = (short *)&alice;/* alice' addr: 10 */
```




```c
printf("%x", alice[0]);   /* prints 48 ('H' in hex) */
printf("%x", alice[1]);   /* prints 65 ('e' in hex) */
```

Pointer

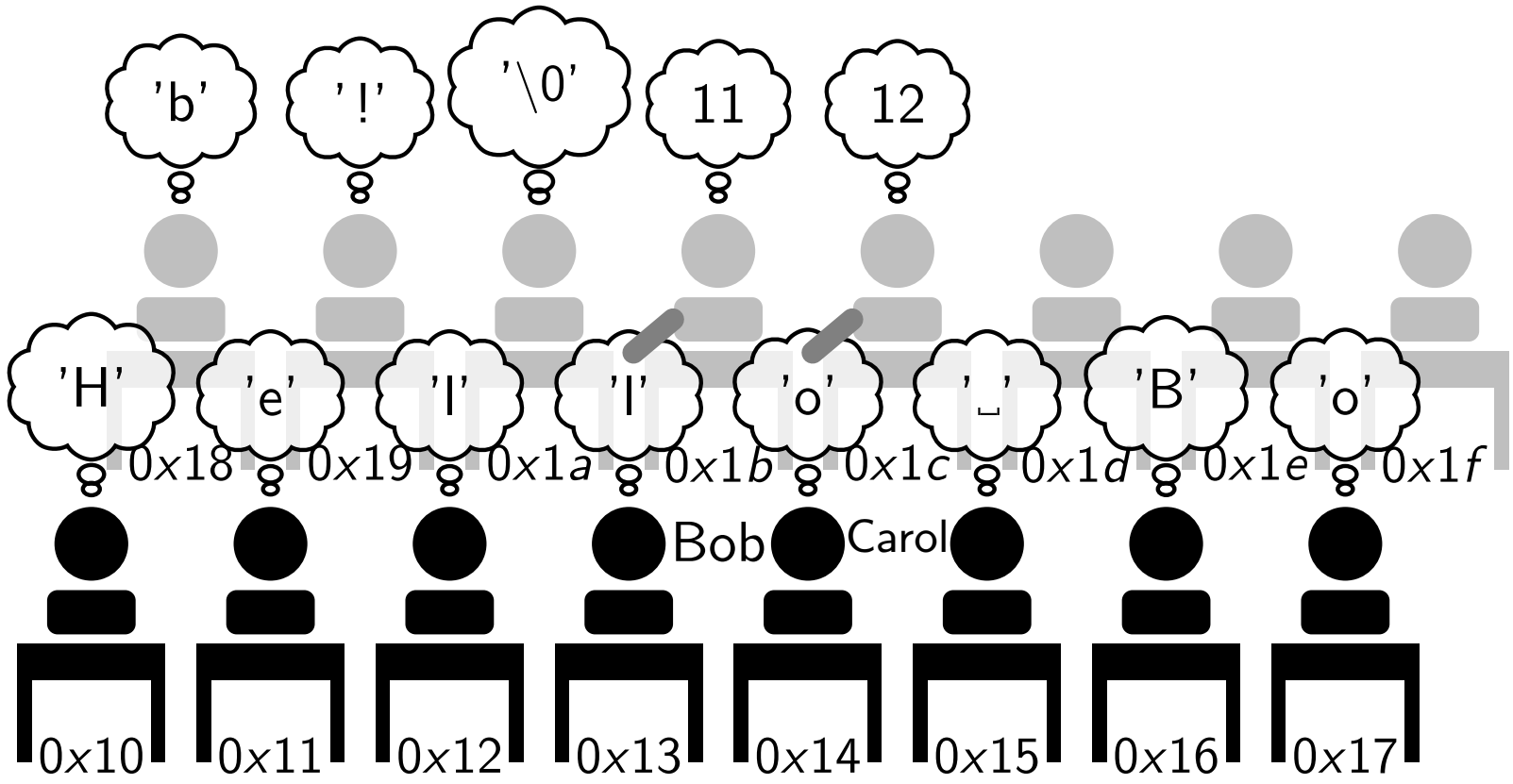
```c
char  alice[] = "Hello Bob!";/* auto \0 terminated */
char  *bob    = &alice;
short *carol  = (short *)&alice;
```




```c
printf("%x", *bob);   /* prints 48 ('H' in hex)     */
printf("%x", *carol); /* prints 6548 (short - 2b!) */
```

Pointer

```c
char  alice[] = "Hello Bob!";/* auto \0 terminated */
char  *bob    = &alice;
short *carol  = (short *)&alice;
```




```c
bob++;   /* next char  + 1 byte */
carol++; /* next short + 2 byte */
```

void

- `void` was introduced in ANSI-C
- Can be described as "empty" data type

It is used as:

- return type for functions which return nothing
- empty parameter list for functions
- pointer with no data type

```
/* void vobject; */
void *pv;
int *pint;
int i;

void foo(void)
{
  pv = &i;              /* OK, but warning */
  pint = (int *)pv;     /* (int *) is necessary in C++,
                         * but not mandatory in C */
}
```

Structs (1/4)

- Combines variables to a logical entity
- The overall size is the sum of the sizes of the single elements
  - Additional size may be used because of alignment
  - The size of the struct can be determined with `sizeof`.
- Elements are accessed using ” .”

```
struct account
{
  char username[32];
  char password[32];
  unsigned int uid;
};

struct account user1 = {"alice", "4l1c3", 1};
struct account user2;

user2.uid = user1.uid + 1;
```

Structs (2/4)

- Tagged Struct:

```
struct account
{
   ...
};
struct account user1, user2;
```

- Untagged Struct:

```
struct
{
   ...
} user1, user2;
```

- Mixed:

```
struct account
{
   ...
} user1, user2;
```

Structs (3/4)

- Since C-99 it is also possible to initialize the variables by their name:

```
struct account
{
  char username[32];
  char password[32];
  unsigned int uid;
};

struct account user1 = {.uid=1,
  .username="alice", .password="4l1c3"};
```

Structs (4/4)

- You can also have pointers that point to structs
- Dereferenced with ->

```
struct account
{
  char username[32];
  char password[32];
  unsigned int uid;
};

struct account user1 = {.uid=1,
  .username="alice", .password="4l1c3"};
struct account *p = &user1;

(*p).uid = 1;
p->uid   = 1; /* easier to read */
```

Singly Linked List

- Nodes are linked via pointers
- Important: *head* has to be known, otherwise parts of the list are lost

```
struct account_node
{
  char username[32];
  char password[32];
  unsigned int uid;
  struct account_node *next;
};

struct account_node user1, user2;
struct account_node *head;

user1.next = &user2; user2.next = NULL;
head = &user1;

// iterate through list
struct account_node *p = head;
while(p != NULL) { p = p->next; }
```

Union

- Unions share the same memory space.
- `sizeof` returns the size of the biggest element
- Only one element is 'active'
- Different views/interpretations of memory content

```
union number
{
  char  c_number;  /* 1 byte */
  short s_number;  /* 2 bytes */
};

union number i;
i.c_number = 0x42;      /* sets 1 byte */
printf("%x", i.c_number); /* prints 42 */
i.s_number = 0x6548;  /* sets 2 bytes */
printf("%x", i.s_number); /* prints 6548 */
printf("%x", i.c_number); /* ?? undefined */
```

enum

- Used to create alias names
- If not specified otherwise, first element gets value 0
- Successive elements' values are incremented by one, if not specified otherwise
- Advantage over `#define`: scope

```
enum [TYPENAME]
{
  IDENTIFIER [= VALUE] [, IDENTIFIER [= VALUE]] *
};

enum boolean {FALSE = 0, TRUE};
enum account {PREMIUM = 1, STANDARD = 2,
              BUSINESS = 4};

enum account account1;
account1 = BUSINESS;
```

Nesting

- Structs and unions can be nested as you like
- Often unions are nested into structs to tag them
- Enums can be used for tagging

```
enum types {A_FLOAT, A_INT};

struct checked_union
{
  enum types type;

  union
  {
    int i;
    float f;
  } value;
};

struct checked_union my_checked_union;

if(my_checked_union.type == A_FLOAT)
  return my_checked_union.value.f;
```

typedef (1/2)

- Possibility to declare (user defined) types

```
/* stdint.h */
...
typedef signed    char    int8_t;
typedef unsigned  char   uint8_t;
typedef signed    int    int16_t;
typedef unsigned  int   uint16_t;
...

#include <stdint.h>
...
uint8_t i;
for(i = 0; i < 10; ++i)
    printf("%u\n", i);
...
```

typedef (2/2)

```
struct account {
  char username[32];
  char password[32];
  unsigned int uid;
};
typedef struct account account_t;
```

...or...

```
typedef struct account {
  char username[32];
  char password[32];
  unsigned int uid;
} account_t;
```

...then we can use account_t like:

```
...
account_t user1 = {"alice", "al1c3", 42};
...
```

Alignment (1/3)

- Data alignment: how is the data organized in the memory?
- Data structure padding: how is the space filled?

Alignment (2/3)

- Before compilation:

```
struct mixed
{
  char  data1;
  short data2;
  int   data3;
  char  data4;
}; /* 8 bytes */
```

- After compilation:

```
struct mixed
{
  char  data1;
  char  padding1[1];
  short data2;
  int   data3;
  char  data4;
  char  padding2[3];
}; /* 12 bytes */
```

Alignment (3/3)

```
struct mixed /* reordered */
{
  int   data3;
  short data2;
  char  data1;
  char  data4;
}; /* 8 bytes */
```

- Best way: order elements of a struct by their size descending
- Problem: Some compilers optimize and reorder the structure elements, others don't → this might end up in accessing wrong memory in, e.g. shared memory.

Function Definition

```
type name(type1 arg1, type2 arg2, ...)
{
  /* code */
}
```

- Attention: `int foo();` != `int foo(void);`
- `int foo();` accepts an undefined length of parameters
- `int foo(void);` does not accept any parameters

Value Parameter

- Value parameters are local parameters within the function, they are not changed

```
void foo(int a)
{
  a = 23;
}

int main(void)
{
  int a = 42;
  foo(a);
  /* the value of a is still 42 */
  return 0;
}
```

Variable Parameter (1/2)

- The function gets a pointer as parameter (as a value)
- The value in the address of the pointer can be changed

```
void foo(int *a)
{
  *a = 23;
}

int main(void)
{
  int a = 42;
  int *b = &a;

  foo(&a);
  /* the value of a is now 23 */
  foo(b);
  /* still 23 */

  return 0;
}
```

Variable Parameter (2/2)

- Arrays are always passed as variable parameters to functions
- Dirty trick: To pass them as a value it is possible to pack them in a `struct`

```
void foo(int *a)
{
  *a = 23;
}

int main(void)
{
  int a[] = { 1, 2, 3, 4, 5, 6 };
  foo(a);
  /* the value of a[0] is now 23 */
  foo(&a[3]);
  /* the value of a[3] is now 23 */
  return 0;
}
```

Pointer and Const

- const is used to declare a variable as read-only
- Is used for variable function parameters where the value must not change

```
char c;
char *const cp = &c;
/* The value to which cp points to can be changed,
 * however, the pointer can't be changed
 */

const char *cp = &c;
/* The value to which cp points to can't be changed,
 * however, the pointer can be changed
 */

char const *cp = &c; /* same as const char *cp */

const char *const cp=&c;
/* The value to which cp points to can't be changed,
 * the pointer can't be changed either
 */
```

Return Values/Pointers (1/2)

- Values as well as pointers can be returned by functions

```
int my_double(int a)
{
  return 2*a;
}

int main(void)
{
  int a;

  a = my_double(5);
  /* value of a is now 10 */

  return 0;
}
```

Return Values/Pointers (2/2)

```
char *first_b(const char *a)
{
  int i;
  for(i = 0; i , strlen(a); ++i)
  {
    if(a[i] == 'b') return &a[i];
  }
  return NULL;
}

int main(void)
{
  char *string1 = "foobar";
  char *string2 = "foofoo";
  char *p = first_b(string1);
  if(p != NULL)
    printf("found a %s at address %x", p, &p); // bar
  return 0;
}
```

Inline

```
inline type name(type1 arg1, type2 arg2, ...)
{
  /* code */
}
```

- According to the standard (from C-99) the code should run as fast as possible (a hint for the compiler)
- The implementation is not mandatory
- Instead of calling the function, the function code is often copied into the code that is calling the function
- Can be ignored

Pointers to Functions

- It is also possible to assign functions to pointers

```
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int main(void)
{
  int (*f)(int, int);
  int ret;

  f = &add;
  ret = f(42,23);
  /* ret == 65 */

  f = sub; /* f = &sub is better */
  ret = (*f)(42,23);
  /* ret == 19 */

  return 0;
}
```

Procedural Programming (1/2)

C is

- a procedural language
- not like Java - it is not OOP
  - You do not have objects
  - There is no inheritance

In C you use procedures/functions/methods that operate on/modify data structures.

Procedural Programming (2/2)

```
typedef struct
{
  char username[32];
  char password[32];
  unsigned int uid;
} account_t;

void acc_init(account_t *account)
{
  /* check username, password */
  /* if everything's successful,
   * create account in database
   * and assign uid in account*/
}

account_t account1 = {"alice", "4l1c3", 0};
acc_init(&account1);
if(account1.uid != 0)
  printf("Account successfully initialized!");
```

Modular Programming

- Increases readability, re-usability and maintainability
- Module is split into header (*.h) and source (*.c) files.

Header Files

- Contains prototypes and constants
- Contains no definitions of functions (implementation is done in *.c files)
- `#include` is used to include modules
  - `#include <account.h>` searches in library path
  - `#include "account.h"` searches in local folder

```
/* account.h */
#ifndef ACCOUNT_H /* include guard */
#define ACCOUNT_H

typedef struct { ... } account_t;

void acc_init(account_t *);
void acc_set_password(account_t *, const char *);

#endif /* ACCOUNT_H */
```

Source Files

- C files in which the functions are implemented
- Functions that are defined with `static` are available only in the current file

```
/* account.c */
#include "account.h"

void acc_init(account_t *account)
{
  /* do stuff here to initialize
   * the account (check dublicates.
   * constraints etc.) */
  /* assign uid if done correctly */
}

void acc_set_password(account_t *account,
                      const char *pw)
{
  /* set password for account */
}
```

Use the module

```c
#include "account.h"

int main(void)
{
  account_t account;
  account.username = "alice";
  account.password = "4l1c3";

  acc_init(&account);
  acc_set_password(&account, "n3wl1c3");

  return 0;
}
```

Compilation

- Projects that consist of several modules are compiled as follows:

```
$ gcc -c account.c # -> account.o
$ gcc -c prog.c    # file that contains main
$ gcc -o prog prog.o account.o
```

Outlook

Material:

- C Programming Language - Kernighan & Ritchie
- C Traps and Pitfalls - Andrew Koenig
- https://en.wikibooks.org/wiki/C_Programming
- https://de.wikibooks.org/wiki/C-Programmierung
- http://www.c-faq.com/